Integración de una Caché de Datos en el Sistema en Chip SweRVolf

Integration of a Data Cache in the SweRVolf System on Chip

Alfonso Carballo Boullosa

Grado ingeniería de computadores

Christian Balbás Sánchez

Grado ingeniería informática

TRABAJO FIN DE GRADO
CURSO 2020-2021

FACULTAD DE INFORMÁTICA

UNIVERSIDAD COMPLUTENSE DE MADRID

DIRECTORES

Christian Tomás Tenllado Van Der Reijden
José Ignacio Gómez Pérez

Resumen

Hoy en día el acceso a memoria limita en gran medida el rendimiento de un procesador, por ello, es muy importante tener una jerarquía de memoria que reduzca en lo posible la latencia de acceso a memoria, siendo especialmente relevante el uso de memorias caché.

SweRVolf es un System on Chip (SOC) de código abierto de Western Digital, que integra un procesador RISC-V de la misma compañía, el SweRV EH1. Este procesador dispone de una caché de instrucciones, pero de datos. En este proyecto proponemos un diseño propio de una caché de datos que incorporaremos en la ruta de datos del Swerv EH1.

En nuestro diseño hemos procurado cuidar el aspecto didáctico, tratando que la caché sea configurable, de modo que el usuario pueda escoger distintas políticas de escritura, el tamaño de la caché o el número de vías en tiempo de compilación.

Además del desarrollo de la caché también realizamos pruebas de verificación para comprobar la corrección de ésta.

Palabras clave

Caché, memoria, procesador, RISC-V, SweRV, SOC.

Abstract

Nowadays memory access greatly limits the performance of a processor, therefore, it is very important to have a memory hierarchy that reduces memory access latency as much as possible, that causes the use of cache memories to become especially relevant.

SweRVolf is an open-source System on Chip from Western Digital, which integrates the SweRV EH1 a RISC-V processor from the same company. This processor has an instruction cache, but no data cache. In this project we will design our own data cache memory and will be incorporated into the Swerv EH1 data path.

Our design is focused to be didactic, ensuring that the cache is configurable, so that the user can choose different writing policies, the size of the cache or the number of paths at compile time.

In addition to the development of the cache, we also carry out verification tests to verify the correctness of the cache.

Keywords

Cache, memory, processor, RISC-V, SweRV, SOC.

Índice de contenidos

Capítulo 1 Introducción

1.1 Antecedentes

A finales del siglo XX aparecen las primeras FPGAs y Open Design Circuits, el primer movimiento que promovió una comunidad de diseño de hardware libre. Esto permitió que diseños electrónicos libres pudieran ser intercambiados. Sin embargo, dada la inexistencia de software libre para diseño electrónico, el crecimiento de la comunidad de hardware libre fue mínimo.

Con el paso del tiempo el concepto de hardware libre ha tomado más importancia y presencia en la comunidad, surgiendo proyectos como RepPap, Arduino y Raspberry Pi.

Uno de los proyectos que mayor impacto ha tenido en la comunidad de hardware libre es RISC-V. Nace en el año 2010, en la universidad de California de Berkeley, con una arquitectura de conjunto de instrucciones (ISA) de tipo RISC (conjunto de instrucciones reducido).

RISC-V permite a empresas apostar por desarrollar procesadores personalizados ajustándose a sus necesidades, reduciendo los costes ya que se evita el pago de gestión digital de derechos (DRM) a otras compañías. Esto le puede proporcionar a Europa una oportunidad para tener independencia de los procesadores desarrollados por las principales potencias mundiales. Actualmente existen una gran cantidad de organizaciones que apuestan por RISC-V, incluso en el caso del Barcelona Supercomputing Center.

Adicionalmente y gracias a la disponibilidad de gran cantidad de documentación técnica sobre RISC-V, está propiciando un cambio en la comunidad educativa. Hasta la fecha el estudio de procesadores era difícil por la inaccesibilidad a información sobre este tipo de hardware, debido a la alta confidencialidad por parte de las empresas punteras. Tanto es así que libros que hasta la fecha estaban basados en procesadores como MIPS y ARM están publicando ediciones de RISC-V.

En este momento existen más de 100 procesadores abiertos publicados en la página oficial de RISC-V [1], con diferentes repertorios y extensiones, publicados por diferentes compañías, como Western Digital, CodaSip, SiFive o UCB BAR, pero

ninguno de éstos dispone de una memoria caché de datos, lo que podría ser muy útil para la comunidad.

1.2 Objetivos

Dado a que ningún procesador RISC-V abierto publicado hasta el momento dispone de caché de datos, decidimos diseñar e implementar una para alguno ya existente. Como ya existe un proyecto educativo basado en el SweRVolf y una adaptación de este SOC para FPGA llamada RVfpga desarrollado por Imagination, es el hardware escogido para este trabajo.

Los objetivos que han determinado el diseño de la caché son los siguientes.

- Mantener el funcionamiento del procesador. Minimizar los cambios fuera de los módulos de la caché para facilitar así una posible adaptación en otros procesadores. Además, compatibilizar nuestra caché con todas las funcionalidades ya existentes en el procesador.
- Parametrización arquitectónica. Posibilidad de configurar la arquitectura de la memoria. Elementos como el número de vías, tamaño de bloque, número de sub-bancos, tamaño de bloque y profundidad de la memoria caché.
- Parametrización de políticas. Posibilidad de configurar las políticas de escritura de la caché. Write allocate o non write allocate y write back o write through.
- Implementación del SOC con la caché en una FPGA Nexys 4 DDR (Artix 7).

Como consideración adicional, queríamos que la extensión del procesador con la caché pudiese ser usado como base para un RISC-V de ámbito industrial.

1.3 Plan de trabajo

El trabajo realizado para este proyecto fue divido en las siguientes fases.

- Investigación. Lectura de la documentación asociada al SOC para comprender la estructura y funcionamiento básico del procesador.
- Instalación de entorno de desarrollo. Familiarización con el software para la simulación, edición, implementación y depuración del procesador.
- Exploración inicial. Simulación del procesador y revisión del código de descripción hardware (HDL) para comprender el flujo de trabajo de la LSU.
- Diseño inicial caché. Con el fin de entender la ruta de datos se diseñó una caché sencilla basada en registros para descubrir las señales e interacciones más relevantes con el procesador.
- Exploración. Con el objetivo de desarrollar un código homogéneo, investigamos la estructura de la caché de instrucciones ya implementada en el procesador. También estudiamos la manera de comunicar la caché de datos, con la memoria principal mediante el bus AXI (interfaz extensible avanzada).
- Desarrollo. Creación de los módulos finales de la caché de datos, basando su estructura en la de la caché de instrucciones.
- Depuración. Detección y corrección de errores producidos en la fase de desarrollo anterior.
- Mejora. Implementación de nuevas políticas, parametrización de la caché y otras funcionalidades no soportadas hasta el momento.
- Pruebas. Probamos el procesador en casos críticos utilizando códigos en ensamblador para forzar posibles errores y corregirlos.
- Benchmarking. Codificamos nuestro benchmark para evaluar la mejora del procesador.
- Implementación. Intentamos sintetizar e implementar nuestro procesador en FPGA, pero sin éxito. No cumplimos los requerimientos de tiempo, para lo cual sería necesaria otra iteración de diseño.

El control de versiones en la fase de exploración se hacía con Google Drive donde subíamos a una carpeta común las referencias, documentos relevantes para el proyecto, así como algunos archivos propios aclarando conceptos importantes sobre el procesador. En la fase de diseño usamos github para el control de versiones del código del procesador, así como todos los archivos necesarios para la simulación de este. Finalmente, para la memoria usamos OneDrive por su buena integración con Word, editor de texto usado para la escritura de ésta.

Capítulo 2 RISC-V y SwervCore EH1

2.1 ISA RISC-V

La arquitectura del procesador (ISA) determina las instrucciones que soporta, su codificación, modos de direccionamiento, elementos visibles al programador, modelo de memoria, excepciones y algunos otros parámetros arquitectónicos. RISC-V es una ISA hardware libre que, a diferencia de la mayoría es modular. Consta de varios repertorios base y una serie de extensiones que nos permiten añadir funcionalidades especificas al conjunto de instrucciones básico[2].

Existen cuatro repertorios de instrucciones básicos:

- RV32I: repertorio de instrucciones base con enteros de 32 bits.
- RV32E: repertorio de instrucciones base (empotrado) con enteros de 32 bits, 16 registros.
- RV64I: repertorio de instrucciones base con enteros de 64 bits.
- RV128I: repertorio de instrucciones base con enteros de 128 bits.

A los que se le pueden añadir las siguientes extensiones:

- M – extensión para multiplicación y división.
- A – extensión para instrucciones atómicas.
- F – extensión para precisión simple FP.
- D - extensión para precisión doble FP.
- C – extensión para instrucciones comprimidas.

2.2 SwervCore EH1

Western Digital ha desarrollado tres procesadores con ISA RV32I, pero diferentes características arquitectónicas como se muestra en la figura 1. Estos son: SweRV EH1, SweRV EH2 y SweRV EL2.

Core Name	RISC-V Type	Pipeline Stages	Threads	Size @ TSMC	CoreMarks/Mhz
SweRV Core EH1	RV32IMC	9- dual issue	Single	.11mm @ 28nm	4.9
SweRV Core EH2	RV32IMC	9- dual issue	Dual	.067 @ 16nm	6.3
SweRV Core EL2	RV32IMC	4- single issue	Single	.023 @ 16nm	3.6

Figura 1 – Tabla de características de los cores desarrollados por Western Digital. Fuente: www.risc-v.org

Existe una adaptación para ser implementada en FPGAs denominada RVfpga. Usa el procesador Swerv EH1 en su versión 1.6 en el SOC SweRVolf y es por lo que basaremos nuestro proyecto en éste[3].

SweRV EH1 es un procesador de 32 bits, superescalar doble issue con una ruta de datos dividida en 9 etapas. Consta de 4 unidades aritmeticológicas (ALUs), organizadas en 2 rutas independientes, I0 e I1 que funcionan de forma paralela. Ambas rutas soportan operaciones aritmeticológicas, I0 también procesa loads y stores e I1 multiplicaciones con una latencia de 3 ciclos[4]. Adicionalmente, dispone de un divisor externo a las rutas de datos con una latencia de 34 ciclos. La ruta de datos del procesador se ilustra en la figura 2.

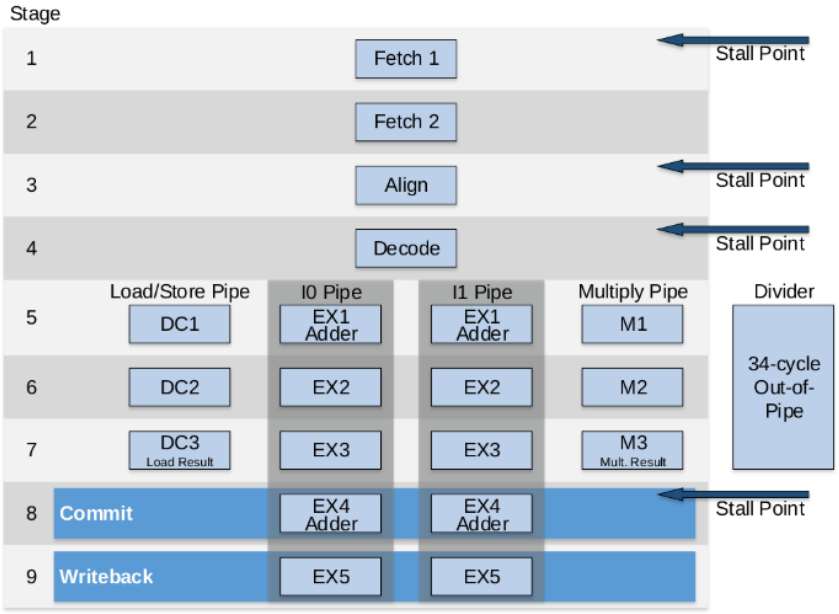

Figura 2 – Etapas del SweRV Core EH1. Fuente: https://github.com/chipsalliance/Cores-SweRV/blob/master/docs/RISC-V_SweRV_EH1_PRM.pdf

Las etapas de la ruta de datos más relevantes para la comunicación con la memoria son:

1. **Decode**: las señales de control y los operandos son generados para el cálculo de la dirección efectiva del load o store.
2. **DC1**: se calcula la dirección final de la operación, se comprueba la región de memoria a donde se dirige y se alinea la dirección. Las diferentes regiones de memoria serán explicadas más adelante.
3. **DC2**: se solicita el dato a la región correspondiente de memoria o se envía al store buffer y se escribirá en memoria unos ciclos más tarde.
4. **DC3**: El dato es leído o escrito de la región correspondiente y es propagado a la siguiente etapa de la ruta de datos.
5. **EX4** o **DC4**: el dato es propagado a la siguiente etapa de la ruta de datos.
6. **EX5** o **DC5**: el valor leído de memoria es escrito en los registros arquitectónicos.

Es las estas etapas explicadas anteriormente es donde debemos centrar nuestro estudio del procesador, ya que es donde intervendrá nuestro controlador de memoria caché.

El control de la ruta de datos requiere el uso de puntos de stall o paradas. Estos son estados en los que una instrucción puede parar de fluir por la ruta de datos, generalmente para esperar los ciclos necesarios para la solución de dependencias. Para que la ruta de datos gestione las paradas correctamente, los huecos que se generan por la parada en un punto son sustituidos por instrucciones burbuja que

será implementada como una no operación (NOP). En nuestro procesador los puntos de parada están situados en los estados 1, 3, 4 y 8, como se puede apreciar en la figura 2.

Las instrucciones para realizar las trasferencias con la memoria son los loads y stores. Existen diferentes tipos de estas instrucciones para transferir datos de diferentes tamaños explicados en la figura 3.

	Sintaxis	**Tamaño**	**Acción**
Load address	*la rd, symbol*	32 bits	Guarda en x[rd] la dirección *symbol*. Donde x[rd] es el valor del registro número rd.
Load byte	*lb rd, offset(rs1)*	8 bits	Lee un byte de la dirección x[rs1]+offset de memoria y lo escribe en x[rd]
Load byte unsigned	*lbu rd, offset(rs1)*	8 bits	Lee un byte de la dirección x[rs1]+offset de memoria y lo escribe en x[rd] extendiendo el resultado con ceros
Load halfword	*lh rd, offset(rs1)*	16 bits	Lee dos bytes de la dirección x[rs1]+offset de memoria y los escribe en x[rd]
Load halfword unsigned	*lhu rd, offset(rs1)*	16 bits	Lee dos bytes de la dirección x[rs1]+offset de memoria y los escribe en x[rd] extendiendo el resultado con ceros
Load word	*lw rd, offset(rs1)*	32 bits	Lee una palabra (4 bytes) desde la dirección x[rs1]+offset y la escribe en x[rd]
Store byte	*sb rs2, ofset(rs1)*	8 bits	Almacena el byte menos significativo del registro x[rs2] en la dirección x[rs1] de memoria
Store halfword	*sh rs2, ofset(rs1)*	16 bits	Almacena los dos bytes menos significativos del

			registro x[rs2] en la dirección x[rs1] de memoria
Store word	*sw rs2, ofset(rs1)*	32 bits	Almacena x[rs2] en la dirección x[rs1] de memoria

Figura 3 - Tipos de instrucciones de memoria del SweRV Core EH1.

Existen dos modos de funcionamiento para las instrucciones de carga de datos.

- **Blocking loads:** se solicita el dato a memoria y la ejecución de la ruta de datos no continúa hasta que el dato solicitado es leído de memoria.
- **Non blocking loads**: el dato se solicita a la memoria y la ejecución de la ruta de datos continúa normalmente si no hay dependencias de datos. Unos ciclos más tarde, cuando la memoria proporciona el dato, este es guardado en el registro correspondiente por una ruta de datos alternativa.

Western Digital ofrece una extensión del SweRV EH1 Core, denominada SweRV EH1 Core Complex, que añade elementos al procesador (véase figura 4), entre los que destacamos:

- DCCM, ICCM: son memorias dedicadas, con rangos de direcciones asignados, que están cercanas al procesador lo que permite ofrecer accesos de baja latencia (1 ciclo). Están destinadas a datos e instrucciones respectivamente e implementan SECDED y ECC (códigos de corrección de errores simple y doble). Ambas memorias se pueden configurar como potencias de 2, desde 4 a 512 KB.
- Caché de instrucciones (ICache): es una memoria caché de instrucciones opcional, de 4 vías asociativa por conjuntos con paridad o ECC, según configuración.
- Bus masters: interfaces de comunicación con el bus AXI, utilizadas para acceso directo a memoria (DMA), accesos a memoria y depuración.

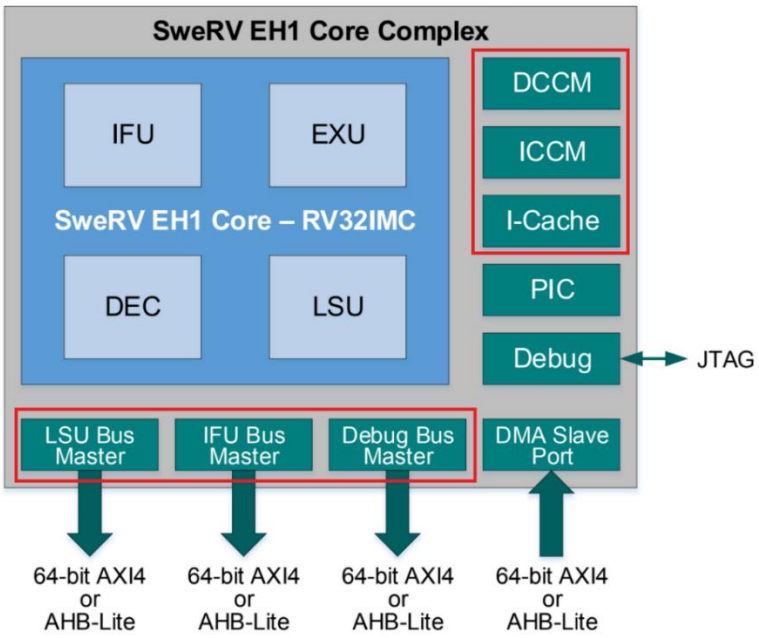

Figura 4 – Componentes SweRV EH1 Core Complex. Fuente: https://github.com/chipsalliance/Cores-SweRV/blob/master/docs/RISC-V_SweRV_EH1_PRM.pdf

Ya que nuestro objetivo es implementar una caché de datos, es especialmente importante entender la estructura de la ICache y su comunicación con memoria principal. Como se muestra en la figura 5, se trata de una caché asociativa por conjuntos con 4 vías de 256 palabras cada una. Las vías están organizadas en 4 sub-bancos entrelazados a nivel de palabra (palabras consecutivas en distintos bancos). La caché utiliza bloques de 64 bytes, formados por 4 entradas consecutivas de cada banco.

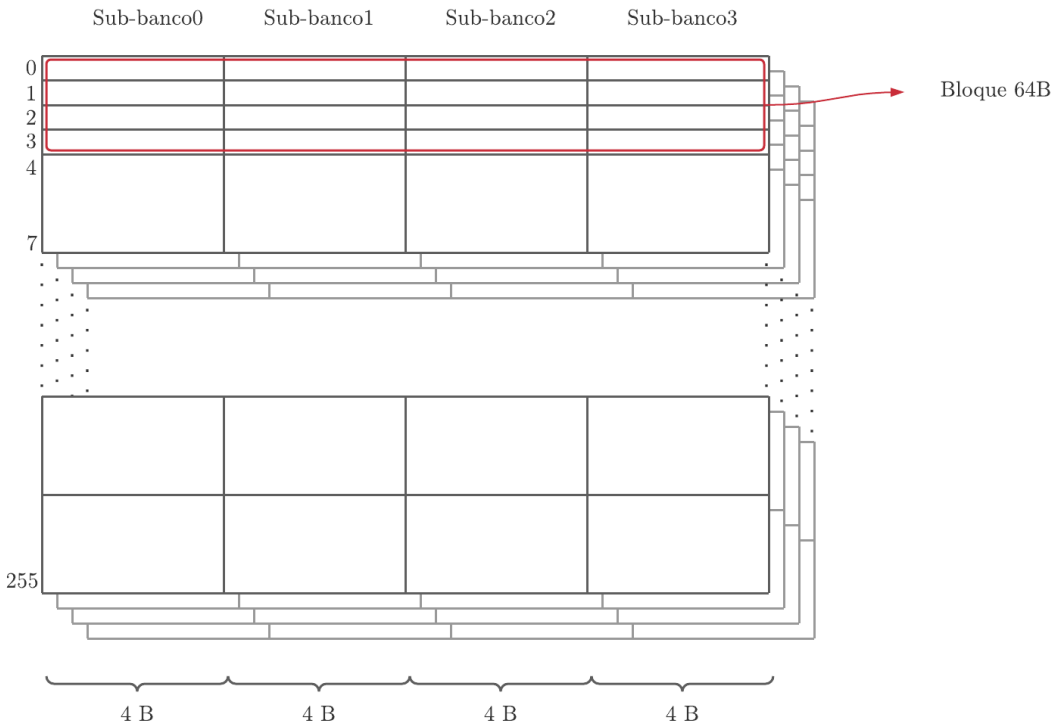

Figura 5 – Estructura de la caché de instrucciones.

La comunicación con los elementos externos al procesador (como la memoria principal) se realiza mediante el uso del bus AXI, que explicaremos en el siguiente apartado. Para la conexión con estos elementos y como se muestra en la parte inferior de la figura 4, el procesador dispone internamente de tres conexiones master al bus: LSU (unidad de load y store), IFU (unidad de fetch) y debug (conexión destinada para depuración).

2.2.1 Bus AXI

La comunicación con la memoria principal se hace a través del bus AXI, por lo que es necesario conocer el protocolo de comunicación.

Como se puede ver en la figura 6, es una interfaz de comunicación paralelo de alto rendimiento semisincrona de alta frecuencia, multi-master y multi-slave. Existen dos extremos de comunicación, uno de ellos es *master o manager* y otro es *slave o subordinate* y están interconectados entre sí. Los *masters* pueden iniciar transferencias de escritura o lectura, mientras que los esclavos están a la espera de nuevas transferencias[5].

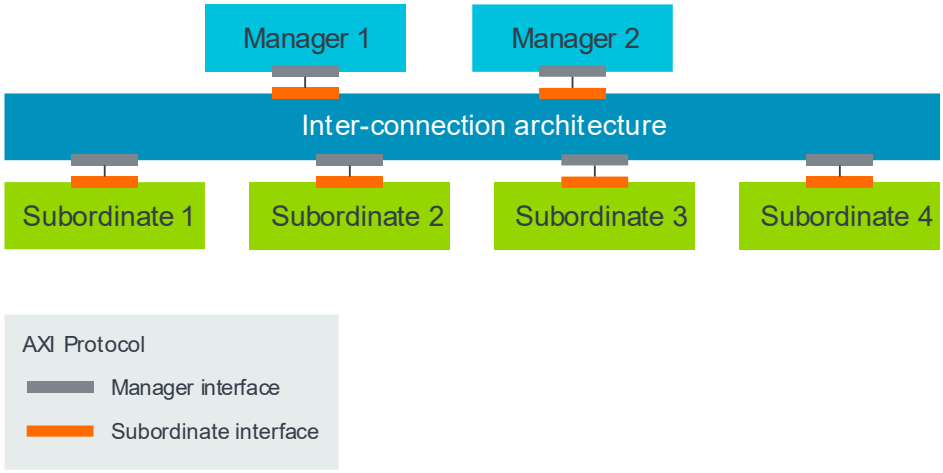

Figura 6 - Interconexión bus AXI [5].

El bus AXI implementa lectura y escritura independientes mostrados en la figura 7. La lectura usa los canales *Read Address* y *Read Data* mientras que la escritura usa *Write Address*, *Write Data* y *Write Response*. Cada canal tiene señales independientes destinadas al envío de señales de control o datos para cada uno de los canales.

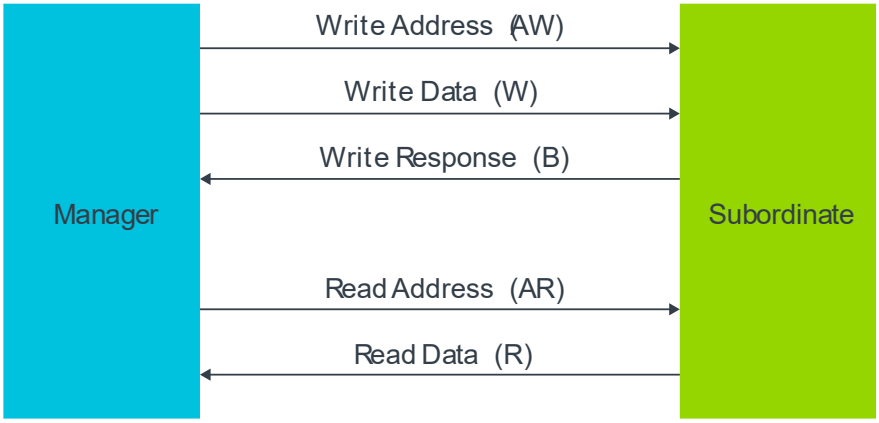

Figura 7 - Comunicación entre Manager y Subordinate[5].

Para los canales mencionado anteriormente se dispone de tres señales específicas destinados al control de la comunicación. Estas son: *valid, ready, last.*

Transacciones

El *handshake* permite iniciar una transacción entre un *master* y un *slave*. Para realizar un *handshake,* y como se muestra en la figura 8, el *master* debe poner a 1 la señal de *valid* y el *slave*, para aceptarlo debe poner a 1 el *ready*, si ya lo estuviese, se mantiene 1 ciclo y el *handshake* se da por completado (véase figura 9). Ambas señales se mantendrán juntas a 1 durante un ciclo y el *handshake* habrá sido completado. Además, mientras que *valid* está a 1, se enviará la información a comunicar al *slave*.

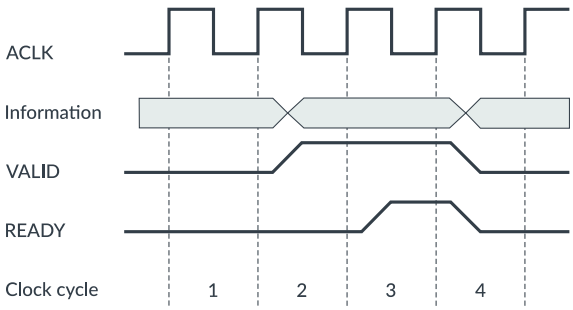

Figura 8 - AXI handshake[5].

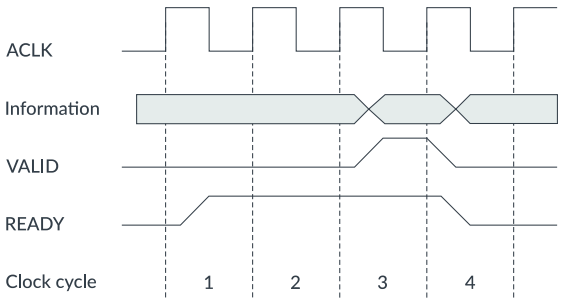

Figura 9 - AXI handshake[5].

El *handshake* anteriormente explicado necesario para las transacciones de lectura y escritura que explicaremos a continuación.

Como se muestra en la figura 10, la transacción de lectura es iniciada por el master con un *handshake* en el canal de *Read Address* donde envía la dirección a leer de memoria. A continuación, el *slave* contesta con otro *handshake* por el canal de *Read Data* con los datos solicitados.

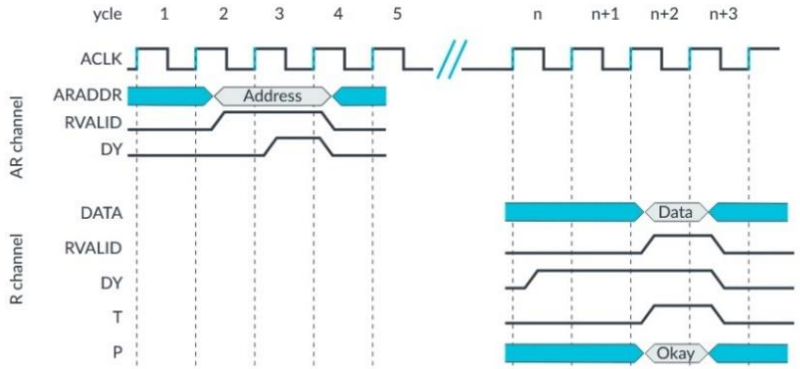

Figura 10 - AXI transacción de lectura[5].

La transacción de escritura se inicia con un *handshake* en el canal de *write address* indicando la dirección en la que se va a escribir, figura 11. Una vez realizado y cuando el *slave* está listo para recibir los datos (*ready* establecido a 1), realiza otro *handshake* mediante canal de *write data* donde el *master* le envía los datos a escribir. Finalmente, el *slave* confirma la recepción de los datos mediante un último *handshake* por el canal de *write response*.

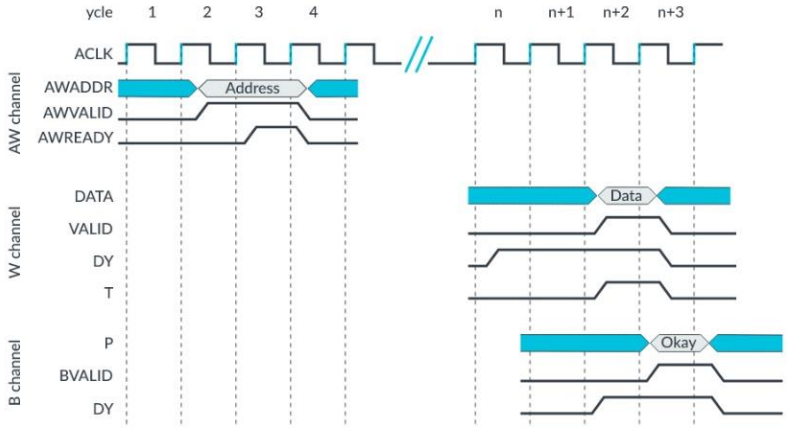

Figura 11 - AXI transacción de escritura[5].

Tanto las transacciones de lectura como escritura permiten el uso de *burst*, que son transacciones que para un único *request*, se envían o solicitan múltiples transferencias de datos.

Capítulo 3 Caché de datos y adaptación

La caché es una memoria colocada entre el procesador y la principal, su función es almacenar los datos de los potenciales accesos futuros obtenidos en base a los accesos realizados. Debido a su arquitectura, tamaño, tecnología (SRAM) y a su cercanía al procesador, los accesos a la misma son muy rápidos[6].

Es necesario definir los siguientes conceptos para comprender el funcionamiento de una caché de datos:

- Bloque. La memoria caché se divide en bloques, todos del mismo tamaño. Cada bloque contiene datos ubicados en posiciones de memoria consecutivas. Es la mínima unidad de transferencias de lectura entre memoria principal y caché.
- Fallo. Ante un acceso a memoria principal, si se determina que el bloque correspondiente no está en la memoria caché, obtendremos un fallo de caché.
- Acierto. Ante un acceso a memoria principal, si el bloque está almacenado en la memoria caché, tendremos un acierto.

La implementación de la memoria caché implica la definición de una serie de políticas:

- Emplazamiento: determina en qué línea de caché se almacena cada bloque de memoria.
- Política de reemplazo. En caso de que todas las vías estén llenas para un conjunto de caché determinado, esta política determina qué bloque se reemplazará.
- Política de escritura. Determina cuándo se actualiza la información en memoria principal. También se determina el comportamiento ante un fallo en memoria caché producido por un store.
- Política de búsqueda de bloques: determina cuándo un bloque es traído de memoria caché. Lo más común es ante un fallo de caché.

Políticas de emplazamiento

- Emplazamiento directo: un bloque de memoria principal solo puede ubicarse en una línea de caché.

- Emplazamiento asociativo: un bloque de memoria principal puede ubicarse en cualquier línea de memoria caché.
- Emplazamiento asociativo por conjuntos: un bloque de memoria principal solo puede alojarse en un conjunto, que puede contener varios bloques, como se aprecia en la figura 12.

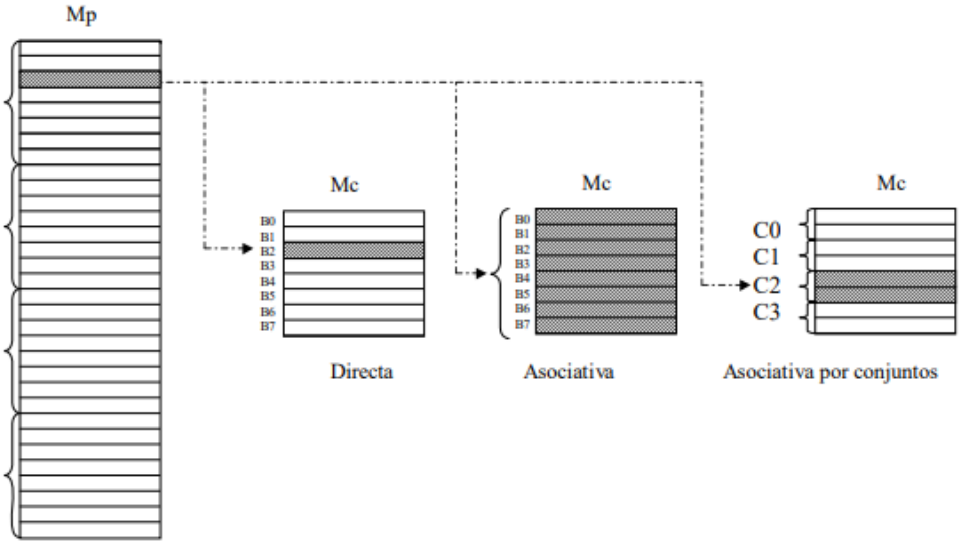

Figura 12 - Tipos de emplazamiento de la memoria caché. Fuente: http://www.fdi.ucm.es/profesor/jjruz/WEB2/Temas/EC6.pdf

El emplazamiento directo, a pesar de ser la opción más rápida y menos costosa a nivel de recursos, presenta problemas ya que varios bloques de memoria principal compiten por una sola línea de memoria caché, pudiendo reemplazar un bloque estando el resto de la memoria caché vacía. Para solventar este problema surgen los emplazamientos asociativos, solucionando el problema de reemplazo de bloques innecesarios, pero siendo mucho más costosas a nivel de hardware y más lentas. Si se combinan las dos ideas, surge el emplazamiento por conjuntos, ofreciendo un equilibrio entre velocidad, coste y eficiencia.

Políticas de reemplazo

En caso de disponer de varias vías por conjunto, cuando todas ellas están llenas y un nuevo bloque va a ser escrito en memoria caché, es necesario determinar qué bloque de ese conjunto va a ser expulsado de la caché. Existen diferentes políticas a utilizar, entre las que destacamos:

- First in first out (FIFO): el bloque sustituido es el que más tiempo ha estado en la caché.

- Last recently used (LRU): el bloque sustituido es el que hace más tiempo que no ha sido utilizado.
- Least Frequently Used (LFU): el bloque sustituido es aquel que ha sido referenciado menos veces.

A priori, podemos pensar que LRU es la política más lógica para implementar en una memoria caché. Sin embargo, al igual que LFU, la implementación es muy costosa a nivel de hardware. Por otro lado, FIFO es más sencilla de implementar y menos costosa a nivel de hardware, pero es menos eficaz.

Políticas de escritura

En las escrituras debemos distinguir dos situaciones, cuando hay un acierto o cuando hay un fallo de caché.

En caso de acierto tenemos dos políticas posibles:

- Write through: todas las operaciones de escritura se realizan en memoria caché y en memoria principal.
- Write back: las operaciones de escritura se hacen sólo en memoria caché. Cuando el bloque modificado es expulsado de la caché, se actualizará la memoria principal.

Para los casos de fallo podemos usar.

- Write allocate: ante un fallo de escritura en memoria principal, el bloque en cuestión es cargado a memoria caché.
- Non write allocate: ante un fallo de escritura en memoria principal, el bloque en cuestión no es cargado a memoria caché.

3.1 Adaptación al SwervCore EH1

La adaptación de la caché de datos con el SwervCore EH1 es un aspecto crítico de la fase de la implementación. Es necesario modificar la ruta de datos del procesador para adaptarla al uso de una memoria caché.

Estos cambios se llevaron a cabo de forma poco intrusiva, modificando lo mínimo posible de la ruta de datos existente, y añadiendo la lógica necesaria en nuestros módulos, permitiendo así la máxima modularidad, por si esta caché quisiese ser implementada en otro procesador. La adaptación de la ruta de datos requiere los siguientes cambios:

- La comunicación con memoria principal es asumida por el controlador de memoria caché y realizada mediante su propio máster del bus AXI.
- Multiplexación de datos en función del origen. El origen puede ser de la caché de datos, la DCCM o memoria principal en casos específicos como acceso directo a memoria (DMA).

Adicionalmente para la integración de la caché, hemos tenido que modificar el comportamiento de las etapas de la ruta de datos de memoria.

- DC1: en el caso de que la dirección sea de la memoria principal, se calcula la dirección final y esta se envía la memoria caché para obtener el tag.
- DC2: como se muestra en el diagrama de la figura 12, una vez obtenido el tag, se compara con el tag generado a partir de la dirección. En caso de hit, se solicita la lectura o escritura del dato a la caché. En caso contrario, se envía la dirección al request buffer para traer el bloque (para los store según política de emplazamiento) y se para la ruta de datos hasta recibir el dato. Además, si el bloque está reemplazando a uno valido y esta dirty, enviamos los datos antiguos al store buffer.

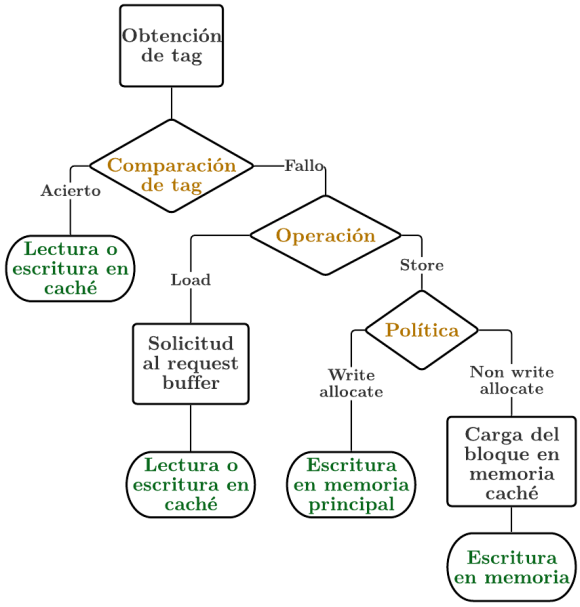

Figura 13 - Diagrama de flujo del comportamiento general de la caché.

- DC3: en caso de haber solicitado una lectura de caché o memoria principal, se devuelven los datos a la ruta de datos.

3.2 Diseño

Los aspectos de diseño que determinan nuestra caché son la arquitectura, los módulos y el comportamiento de estos. A continuación, explicaremos cada uno de ellos.

3.2.1 Arquitectura

Como ya hemos explicado anteriormente, una parte importante de nuestro proyecto era la parametrización de la caché, para ello se han configurado directivas de compilación para tener una arquitectura de la memoria variable. Esto hace que el usuario pueda escoger parámetros físicos de la misma.

- Numero de vías: se admiten configuraciones de 1, 2 y 4 vías.
- Profundidad: llamamos profundidad de la caché al número de palabras que contendrá cada sub-banco. Se admiten 256 y 512.

Las configuraciones anteriormente explicadas son aquellas en las cuales se garantiza el correcto funcionamiento. Se pueden usar otros valores, aunque el comportamiento puede no ser el esperado ya que en procesador no ha sido probado con esas configuraciones.

Por otra parte, el tamaño de bloque (16 palabra de 32 bits y 2 bits adicionales de paridad) y numero de sub-bancos (4) son iguales a los de la caché de instrucciones y no parametrizables.

Según la profundidad escogida, el direccionamiento de la caché cambia. En la figura 14 se muestra el direccionamiento para profundidad de 256.

Direccionamiento con una profundidad de 256

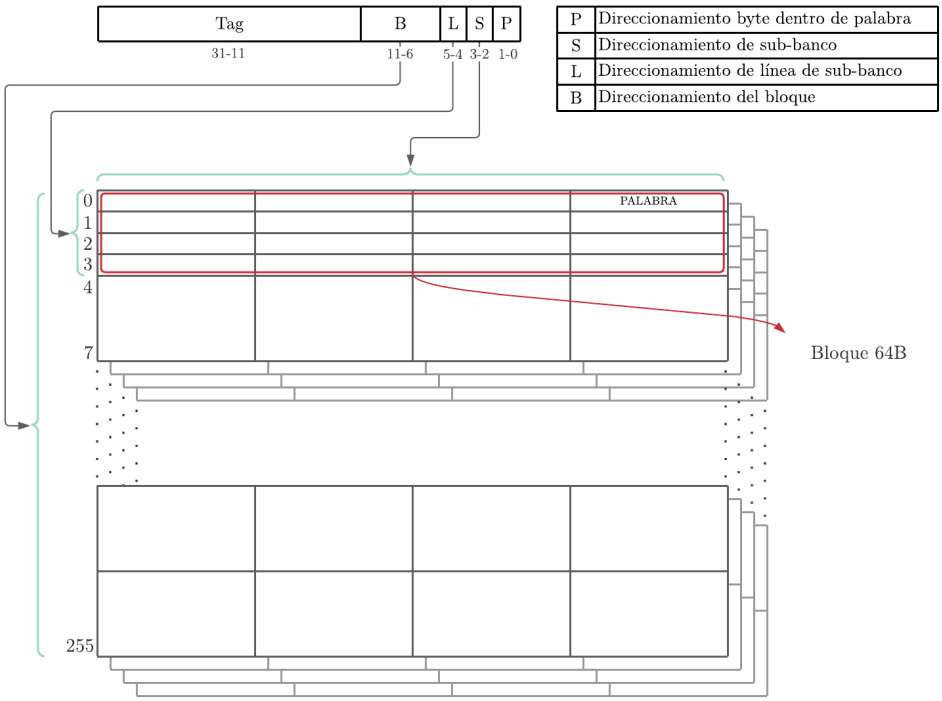

Figura 14 - Direccionamiento caché de datos. Profundidad = 256.

3.2.2 Módulos

Entre los componentes diferenciaremos los módulos físicos, aquellos que a nivel de hardware se muestran como un componente diferente en un archivo independiente. Los módulos lógicos son aquellas partes de los módulos físicos que realizan una función específica sin estar especificado en un archivo independiente.

Módulos físicos

Los módulos físicos de la caché de datos son el controlador de la memoria (especificado en el archivo lsu_dc_mem_ctl.sv) y la memoria (en el archivo lsu_dc_mem.sv).

El controlador de la memoria se encarga de la conexión de la caché con la ruta de datos, así como de la generación de las señales de control. Asume las transferencias de memoria entre memoria caché y principal, para ello interconecta y arbitra los módulos lógicos que explicaremos en el siguiente apartado. Forma parte de la unidad de loads y stores unit (LSU) en la jerarquía de la figura 15.

La memoria se encarga de almacenar los datos, además de una lógica muy básica de comparación para los aciertos. Está dividida en dos sub-módulos, el del tag (encargado del almacén y la comparación de los tags) y el de datos (encargado del almacén de los bloques de datos, bits de paridad y la selección de los datos leídos. Es un módulo de "mem" como se muestra en la figura 15.

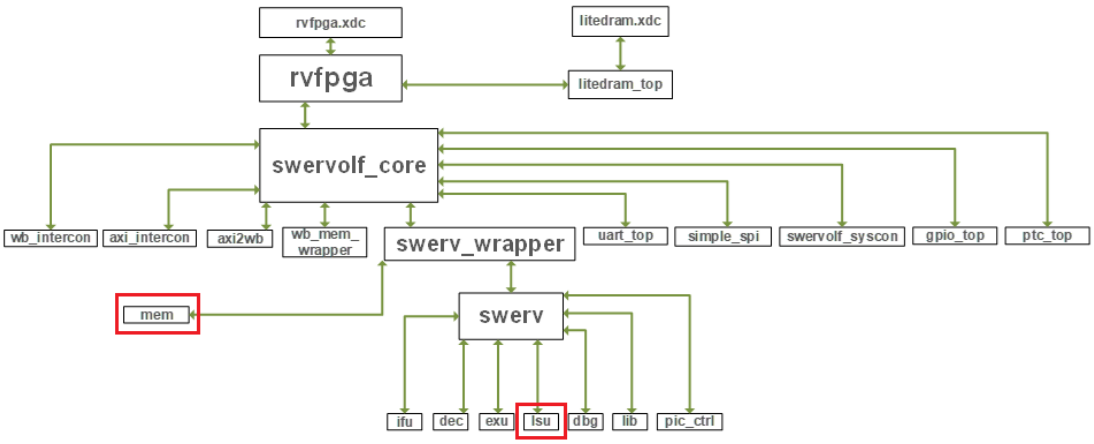

Figura 15 - Diagrama de interconexión del SweRV core EH1 (RVfpga). Fuente: https://university.imgtec.com/teaching-download/

Módulos lógicos

Se usan flags para almacenar el estado de los bloques almacenados en memoria.

- Valid: bit que indica si un bloque tiene al menos una palabra válida en él. Una palabra es válida cuando ha sido escrita por primera vez.
- Wait: bit que indica si una o más palabras de un bloque están a la espera de ser escritas desde memoria principal.
- Dirty: bit que indica si los datos han sido modificados solo en memoria caché. Por lo tanto, este bit solo es necesario, y por lo cual definido, si se usa write back. Disponemos de 8 bits por bloque para reducir el tráfico en el bus, así como el número de escrituras en memoria principal.

Las lecturas y escrituras en memoria principal se hacen mediante el uso de dos buffers.

Request buffer (RDC), es el encargado de hacer las solicitudes de datos a memoria principal mediante el AXI. Para ello, implementamos una serie de registros que guardan la información necesaria para hacer las solicitudes cuando sea posible. Como la implementación del bus AXI en este procesador no soporta la funcionalidad de burst, se ha implementado una máquina de estados (véase figura 16) encargada de realizar las solicitudes de las direcciones de memoria asociadas a cada uno de los bloques de forma secuencial.

- RDC_IDLE: estado de espera, usado mientras el buffer está vacío.
- FR_REQ: se realizan las solicitudes a las direcciones de todos los elementos del bloque de forma secuencial. Para reducir la espera, la primera solicitud es la de la dirección del load que provoco el fallo en caché.
- FR_WAIT: espera la recepción de todos los datos del bloque.

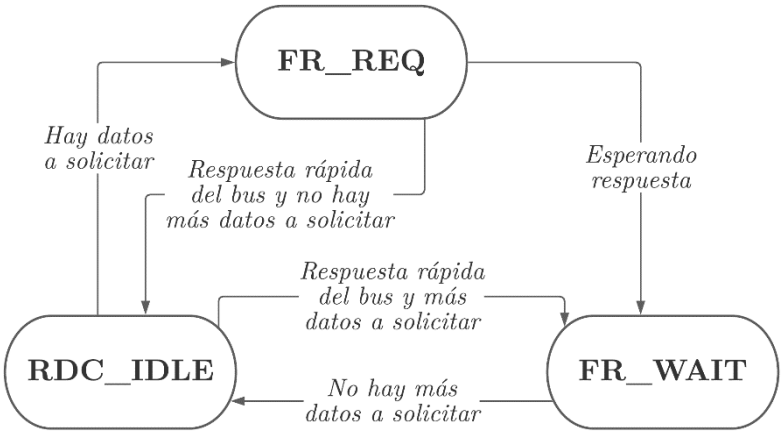

Figura 16 - Diagrama de estados del request buffer.

Store buffer (WDC), es el encargado de almacenar los datos en memoria principal mediante el AXI. El comportamiento de este buffer es dependiente de la política de escritura en memoria principal elegida.

Con write through el buffer tiene una estructura similar al RDC. Para ello, usamos la siguiente máquina de estados (véase figura 17).

- IDLE: estado de espera usado mientras el buffer está vacío.
- SND_ADDR: indica la dirección en la que se quieren escribir los datos.
- SND_DATA: envía los datos para la dirección enviada anteriormente.
- W_RESP: espera la confirmación de recepción de datos por parte de la memoria.
- NO_RESP: controla una falta de respuesta por parte del bus.

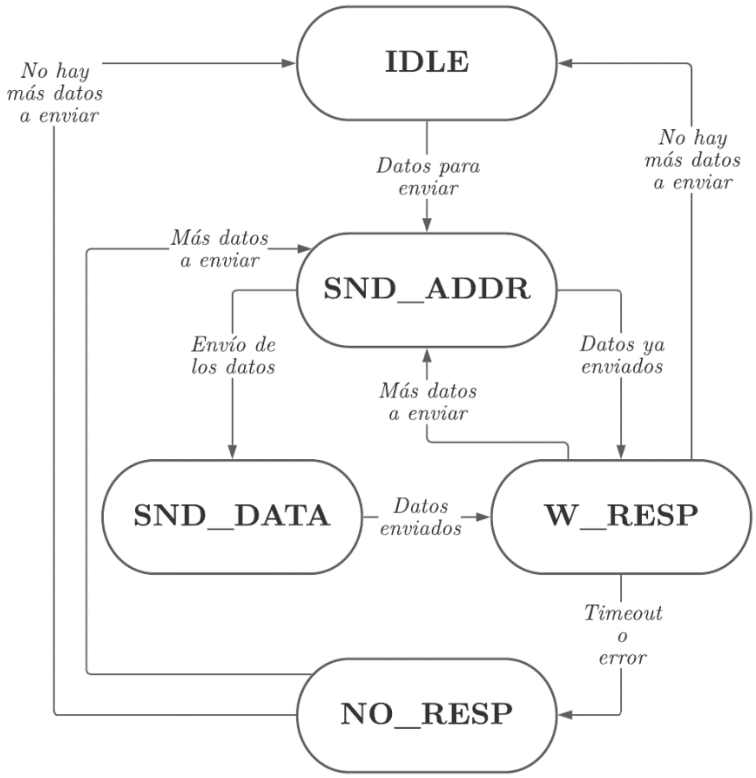

Figura 17 - Diagrama de estados del store buffer con write through.

Con write back la estructura del buffer es ligeramente distinta permitiendo almacenar bloques enteros. En caso de que la entrada vaya a ser una única palabra, esta ocupará el primer lugar en un bloque del buffer y se hará una única solicitud de escritura. Para bloques enteros se revisa el bloque palabra por palabra y, en caso de que el bit de dirty esté activado, se enviara una solicitud de escritura a memoria. Para ello, modificamos la máquina de estados anteriormente descrita añadiendo un nuevo estado (véase figura 18).

- IS_DIRTY: evalúa si la palabra a escribir es dirty. Determinando si se escribe en memoria principal o no.

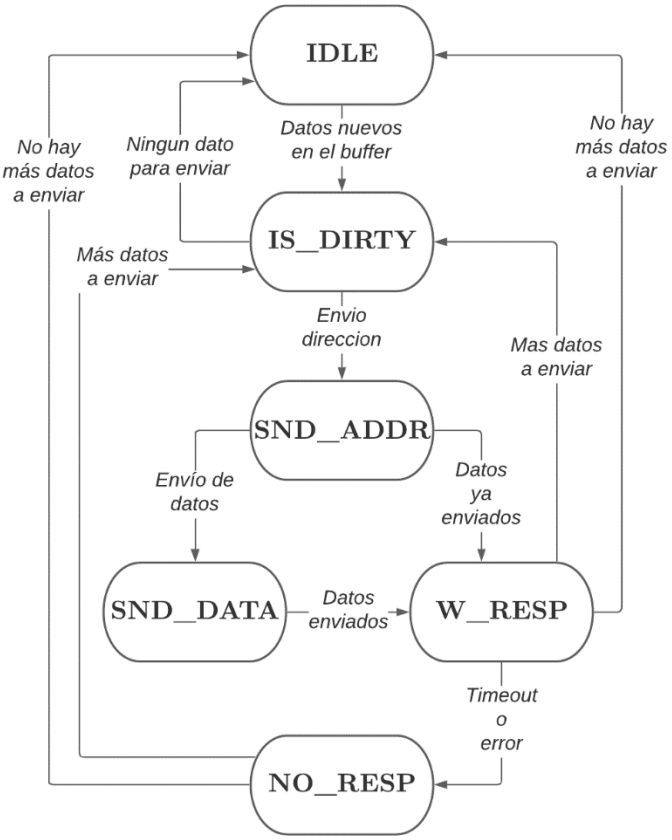

Figura 18 - Diagrama de estados del store buffer con write back.

Las dependencias de datos entre instrucciones son asumidas por otras etapas de la ruta de datos, lo que hace que no tengan que ser gestionadas por el controlador de la caché. Sin embargo, pueden surgir otros riesgos debidas al uso de buffers. Para solucionar este problema, hacemos una parada en la ruta de datos.

La **gestión de paradas** por riesgos internos del controlador de la caché es realizada por el módulo de freeze, encargado de parar la ruta de loads y stores. La parada puede ser causado por diferentes motivos, cada uno identificado por un número.

0. Store buffer (WDC) está lleno.
1. Request buffer (RDC) está lleno.
2. Fallo en caché, esperamos a recibir el dato desde la memoria.
3. Acierto en caché, pero el bit de wait del bloque está activado.
4. La dirección solicitada se encuentra en un bloque pendiente de ser traído de memoria (en el RDC).
5. La dirección solicitada se encuentra en un bloque pendiente de ser escrito en memoria (en el WDC).
6. La instrucción es un store de un tamaño menor a una palabra.

La gestión de paradas se realiza con dos registros. Uno de ellos, almacena el motivo de la parada como una serie de flags basados en la causa o causas que generan la parada (de las mencionadas anteriormente). En otro registro se almacenan las causas que reanudan la ejecución. Cuando los dos registros tengan el mismo valor, se reanuda la ejecución.

Por ejemplo: si el WDC y el RDC están llenos, se almacenará en el registro que causa esta situación. Cuando pare de estar lleno alguno de los dos, se almacenará en el registro de unfreeze y cuando el segundo también tenga algún hueco libre, se reanudará la ejecución.

La **política de reemplazo** de la caché FIFO, es decir, que el primer bloque en entrar en una línea de caché será el primero en ser reemplazado. Para implementar esta política se utiliza un contador, que se incrementa al escribir un bloque, por línea de caché en el que almacenamos la siguiente línea que va a ser reemplazada. Esta política se utiliza en caso de tener varias vías en la caché, ya que con emplazamiento directo solo hay una vía.

3.2.3 Políticas escritura

Las políticas de escritura de la memoria son configurables mediante directivas de compilación que explicaremos en el apartado de parametrización.

Ante un fallo en caché

- Write allocate: el bloque es cargado en memoria caché. El dato es escrito en memoria principal, y se solicita al request buffer la carga del bloque en caché (con el dato ya actualizado).
- Non write allocate: el bloque no es cargado en memoria caché. Se escribe la palabra en memoria principal directamente.

Ante un hit en caché

- Write through: el store se realiza en memoria principal y caché. Se envía la palabra a escribir al store buffer, además de escribirlo en la caché.
- Write back: en las escrituras los datos únicamente son actualizados en caché. Cuando el bloque se reemplace, se escribirán la copia actualizada desde la caché en memoria principal. El bloque es enviado con los bits de dirty (8 por bloque) correspondientes al store buffer, que se encargará de escribir las palabras marcadas como modificadas en memoria.

3.2.4 Manual de configuración

Tanto la arquitectura como las políticas de escritura de la memoria caché son configurables por parte del usuario, para ello se han definido las directivas de compilación que se muestran en la figura 19. Se permiten las configuraciones arquitectónicas mencionadas en el apartado 3.2.1 y selección de políticas de escritura de entre las explicadas en el punto 3.2.3.

Estas directivas se encuentran en el archivo *"/src/SweRVolfSoC/OtherSources/swervolf-swerv_default_config_0.7/configs/snapshots/default/common_defines.vh"*.

Configuración	Parámetros
Habilitar caché	RV_DC_ENABLE deberá ser definido
Habilitar write-back	RV_DC_WB_POLICY_ENABLE deberá ser definido
Habilitar write-through	RV_DC_WB_POLICY_ENABLE no deberá ser definido
Habilitar write-allocate	RV_DC_WRITE_ALLOCATE_ENABLE deberá ser definido.
Habilitar non-write-allocate	RV_DC_WRITE_ALLOCATE_ENABLE no deberá ser definido.
Profundidad = 256	RV_DC_NONE_MUL deberá ser definido.
Profundidad > 256	RV_DC_NONE_MUL deberá ser definido. RV_DC_DEPTH en la línea 14 establecer valor deseado
Número de vías = 1	RV_DC_DIRECT_EMPLACEMENT deberá ser definido.
Número de vías > 1	RV_DC_NUM_WAYS en la línea 20 establecer valor deseado.

Figura 19 - Guía de configuración de la caché.

Para que cualquier cambio de los anteriormente descritos sean efectivos es necesario recompilar el procesador.

Capítulo 4 Validación y benchmarking

Validar la caché requiere realizar las siguientes etapas: depuración (corregir errores) y testing (comprobar el funcionamiento correcto de la caché). Adicionalmente realizaremos una fase de benchmarking para medir la mejora de rendimiento.

4.1 Desarrollo y depuración

La depuración requiere una serie de herramientas. De entre las que podíamos escoger hemos optado, en la medida de lo posible, por utilizar aquellas que son de código abierto o gratuitas para que cualquiera que lo desee pueda replicar nuestro flujo de trabajo.

- Verilator: utilizado para generar la traza de la simulación.
- Vscode + PlatformIO: usado para la depuración de pequeños códigos sobre la FPGA y para la simulación. También usado para la escritura del código ensamblador de los microbenchmarks probados.
- Gtkwave: herramienta gráfica para la visualización de trazas. Usado para ver el valor de señales en simulación, muy útil para la corrección de errores.
- Sigasi: opcional ya que hay multitud de IDEs para código HDL, pero este nos permitía ver en formato de bloques los componentes, lo que facilitó mucho la fase de exploración del código.
- Vivado: usado para la síntesis, implementación y generación del bitstream para la ejecución en FPGA.

Con el objetivo de encontrar errores en el funcionamiento de la memoria caché, empleamos códigos en ensamblador poniendo a prueba el uso de esta en situaciones generales y en casos críticos donde la probabilidad de fallo era mayor. Todas las pruebas fueron realizadas para todas las configuraciones arquitectónicas soportadas y con cada una de las políticas de escritura.

4.2 Testing

A continuación, comprobaremos si el comportamiento de la caché para diferentes situaciones mediante el uso de breves códigos en lenguaje ensamblador. Estas pruebas serán realizadas con diferentes políticas de escritura, así como casos concretos que consideramos relevante probar. Para facilitar la explicación, así como los códigos de prueba para diferentes casos, usaremos una caché de emplazamiento directo con una profundidad de 256.

Para comprobar si el funcionamiento es el esperado para cada una de las pruebas, simulamos los códigos en ensamblador sobre el procesador.

4.2.1 Accesos desalineados

Con el objetivo de comprobar el funcionamiento de los accesos desalineados, se han realizado loads a direcciones de memoria no alineadas a la palabra en los dos casos siguientes:

1. Acceso a un dato desalineado cuyo rango de direcciones esté comprendido dentro de una sola línea de sub-banco. En este caso el funcionamiento de los accesos es el esperado. Representado en verde en la figura 20.
2. Acceso a un dato desalineado cuyo rango de direcciones esté comprendido en dos líneas de sub-bancos distintas. En este caso el funcionamiento no es el esperado y los datos obtenidos son incorrectos. Representado en rojo en la figura 20.

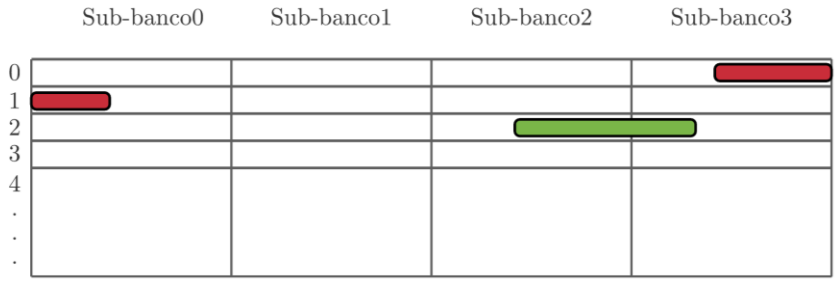

Figura 20 - Tipos de accesos desalineados.

4.2.2 Implementación en FPGA

Después de diseñar la caché, se trató de implementar en el SOC en la FPGA, pero debido a la falta de planificación en el diseño, creamos nuevos caminos críticos que tuvieron un impacto muy negativo en el rendimiento del procesador. Este impacto se puede apreciar en la figura 21.

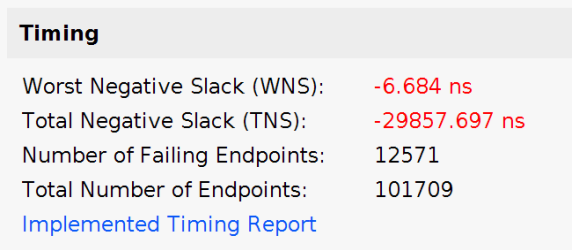

Figura 21 - Timing del proyecto implementado.

Dado que esta violación de tiempo es elevada, la solución requiere un rediseño importante de la caché, pero no fue posible por falta de tiempo para llevar a cabo los cambios requeridos para solventar el problema.

Como solución a este problema, podríamos disminuir la frecuencia de reloj, pero el rendimiento de todo el procesador disminuiría de forma importante, haciendo que la implementación de una caché no tuviese sentido. Por otra parte, podríamos segmentar la caché, esto complicaría un poco la ruta de datos y control, pero con un buen diseño permitiría cumplir los requerimientos de tiempo.

4.2.3 Write back

El objetivo de esta prueba es comprobar el correcto funcionamiento de write back. Para ello, es necesario guardar un bloque en la memoria caché, modificarlo y después cargar un bloque al que le corresponda el mismo conjunto y de esta manera provocar la actualización en memoria principal del bloque modificado. En la figura 22, se puede ver las instrucciones en ensamblador necesarias para realizar la prueba.

```
80      nop
81      nop
82      nop
83      la t1, A            # cargamos en t1 la direccion de A
84      lw t3, 0(t1)        # cargamos el bloque en cache
85      sw t2, 0(t1)        # modificamos un dato del bloque
86      add t1, t1, t5      # direccionamos posicion memoria con diferente tag
87      lw t3, 0(t1)        # reemplazamos el bloque y traemos el nuevo a memoria
88      sw t2, 0(t1)
89      nop
90      nop
91      nop
```

Figura 22 - Código empleado para el ejemplo.

El comportamiento es el esperado ya que los cambios producidos por el store de la línea 85 se actualizan en memoria principal cuando el bloque es sustituido.

4.2.4 Write through

En este caso se ejecuta el mismo código de la figura 22, así podremos apreciar las diferencias del comportamiento entre write through y write back, además de comprobar el funcionamiento correcto. En este caso las escrituras provocadas por las instrucciones de las líneas 85 y 88, se llevan a cabo en memoria principal directamente. Si los bloques implicados en la operación de escritura se encuentran en memoria caché, también se actualizará el dato en ella.

El resultado obtenido en la prueba es el esperado, se pudo apreciar cómo las escrituras se realizan en memoria principal directamente, y en memoria caché en caso de que el bloque implicado esté almacenado en ella.

4.2.5 Write allocate

Para poder comprobar el correcto funcionamiento de write allocate, realizamos dos escrituras consecutivas a la misma dirección (como se muestra en la figura 23). El primer store provocara la carga del bloque en memoria caché, esto hará que la segunda escritura provoque un acierto en caché.

```
75      nop
76      nop
77      nop
78      la t1, A            # cargamos en t1 la direccion de A
79      sw t2, 0(t1)        # guardamos t2 (0x28) en A
80      lw t3, 0(t1)        # cargamos el nuevo valor
81      nop
82      nop
83      nop
```

Figura 23 - Código empleado para el ejemplo.

El comportamiento observado en simulación es el esperado ya que el primer store para la ruta de datos mientras el bloque está siendo cargado y cuando finaliza, el segundo store provoca un acierto en memoria caché.

4.2.6 Non write allocate

En esta prueba el código es el mismo que en la anterior (figura 23). En este caso la primera no provoca la carga del bloque en memoria caché, haciendo que el segundo store no provoque un acierto en caché. En ambas escrituras, es necesario leer de memoria principal el dato, para lo cual la ruta de datos se parará 2 veces.

4.2.7 Dependencias

Las dependencias de datos no requieren un tratamiento especial por parte del controlador de la caché, ya que la gestión de éstas se realiza en otras etapas de la ruta de datos anteriores. En el caso de las instrucciones de carga de memoria, si el bloque no está en memoria caché, debemos parar la ruta de datos hasta obtener el dato que nos interesa. Lo anteriormente mencionado se ha probado con el código en ensamblador de la figura 24.

Inicialmente se carga en el registro t1 una dirección de memoria, el contenido de esta dirección es un dato que será cargado en la instrucción siguiente. Una instrucción después de la carga del dato en el registro t3, se realiza una operación que tiene como operando este registro.

```
76    nop
77    nop
78    nop                 # t5 = 0x00001000
79    la t1, A
80    lw t3, 0(t1)        # cargamos el bloque en cache
81    add t2, t3, t5      # realizamos operacion dependiente
82    nop
83    nop
84    nop
```

Figura 24 - Código empleado para el ejemplo.

El resultado de la suma es el esperado, esto es debido a que la gestión de las dependencias de datos se realiza correctamente.

4.2.8 Paradas en la ruta de datos

A pesar de que, como hemos visto en el ejemplo anterior, las dependencias de datos sean resueltas por la propia ruta de datos, los riesgos creados por el uso de buffers y solicitudes propias al bus tienen que ser resueltos por el controlador de caché. Como ya hemos explicado anteriormente, cada motivo de parada tiene un numero de identificador asignado que será guardado en el registro de origen de parada.

A continuación, mostraremos situaciones donde tenemos que parar la ruta de datos por riesgos creados por el controlador de memoria caché.

Instrucción en el store buffer.

Como se muestra en la figura 25, para realizar esta prueba debemos rellenar el store buffer (WDC) con algunos datos (que van a ser guardados en memoria). Se hace un load a una de las direcciones de los datos del WDC. Esto provoca una parada ya que, para mantener el orden de escritura y lectura, el dato del WDC debe ser escrito primero en memoria. Cuando sea escrito en memoria, la ruta de datos continua su flujo habitual y el dato es cargado desde memoria principal.

```
83    nop
84    nop
85    nop
86    la t1, A
87    sw t2, 0(t1)
88    sw t2, 0(t1)
89    sw t2, 0(t1)
90    sw t2, 0(t1)
91    sw t2, 0(t1)
92    sw t2, 0(t1)
93    sw t2, 0(t1)
94    sw t2, 0(t1)
95    sw t2, 0(t1)
96    sw t2, 0(t1)
97    lw t3, 0(t1)
98    nop
99    nop
100   nop
```

Figura 25 – Código empleado para el ejemplo.

Tras simular este código en el prosador con la caché pudimos ver que el comportamiento de este es el esperado y tal y como se describe en el párrafo anterior.

Instrucción en el request buffer.

De manera similar al ejemplo anterior, y como se muestra en la figura 26 realizamos un load cargando la dirección del bloque a leer de memoria en el load buffer (RDC). Se hace un store a la misma dirección que está provocando la carga de este bloque, como el dato está aún pendiente de ser leído de memoria, la ruta de datos se parara hasta que la totalidad del bloque sea almacenado en memoria caché y posteriormente pueda ser modificado por el store en caché (y memoria principal según política).

```
81    nop
82    nop
83    nop
84    la t1, A
85    lw t3, 0(t1)
86    sw t2, 0(t1)
87    lw t3, 0(t1)
88    nop
89    nop
90    nop
```

Figura 26 – Código empleado para el ejemplo.

El comportamiento del procesador en este escenario es el esperado y tal y como se explica en el párrafo anterior.

Acceso a un bloque con wait.

La carga de los bloques en memoria caché no se realiza de manera inmediata, sino que requieren una gran cantidad de ciclos, por lo que si se hace un acceso a una dirección de un bloque mientras éste está siendo cargado desde memoria principal, es necesario parar la ruta de datos hasta que el bloque se encuentre en la memoria caché en su totalidad. Para realizar esta prueba hemos utilizado el código en ensamblador de la figura 27 ya que se hacen dos accesos a memoria consecutivos, y el primero de ellos provocará un fallo de memoria caché y hará que se cargue en caché. El segundo acceso provocará un acceso antes de que se complete la carga del bloque, por lo que se parará la ruta de datos. Una vez que se reanuda la ejecución, la instrucción de la línea 87 propiciará un acierto en caché.

```
82      nop
83      nop
84      nop
85      la t1, A
86      lw t3, 0(t1)
87      lw t3, 4(t1)
88      nop
89      nop
90      nop
```

Figura 27 - Código empleado para el ejemplo.

El resultado obtenido fue el esperado, ya que se comportó como se ha descrito en el párrafo anterior.

Store de menos de 1 palabra

Para hacer pruebas de stores de menos de una palabra usaremos el código que se muestra en la figura 28. Haremos que el bloque sea guardado en memoria caché mediante el load de la línea 86, una vez el bloque haya sido guardado en caché hacemos un store de media palabra sobre una dirección del mismo. Dada la tecnología de nuestra caché, solo podemos leer y escribir palabras y no bytes ni medias palabras. Por ello el store provoca una parada de un ciclo en la ruta de datos donde se leerá la palabra donde se escribirá nuestro dato, se combinará con él y se guardará la palabra completa en memoria caché. Finalmente realizamos un último load para comprobar que el dato ha sido modificado correctamente en la caché.

```
82      nop
83      nop
84      nop
85      la t1, A
86      lw t3, 0(t1)
87      sh t2, 0(t1)
88      lw t3, 0(t1)
89      nop
90      nop
91      nop
```

Figura 28 - Código empleado para el ejemplo.

En nuestro caso el funcionamiento es el esperado ya que, se para la ruta de datos durante un ciclo para realizar la lectura de la palabra y el dato obtenido en t3 después del load de comprobación de la línea 88 es el esperado.

Cabe destacar que, en esta prueba, y para evitar repeticiones innecesarias, solo mostramos un store de media palabra, aunque los store de bytes funcionan de forma similar y han sido probados de la misma manera.

Request buffer o store buffer lleno

En la fase de diseño se contempló la posibilidad de que alguno de los buffers se llenase impidiendo así el uso de estos, por lo que se decidió implementar una parada de la ruta de datos cuando esto sucediese. No obstante, no ha podido ser probado durante la simulación, ya que la lectura y las escrituras son más rápidas que los datos entran en los buffers.

4.2.9 Comprobación de hits

En los casos anteriores se han podido comprobar el correcto funcionamiento de la caché en casos específicos. A continuación, se mostrará diferentes pruebas realizadas con códigos en ensamblador propios basados en la filosofía de membench, excitando la caché. Estas pruebas nos permiten comprobar el número de aciertos desde el comienzo de la simulación, para ello se ha implementado un contador hardware que se incrementa ante un acierto de la caché.

Se han hecho pruebas con 3 códigos diferentes.

- Código 1: 6 repeticiones de 100 accesos a bloques consecutivos. Genera 600 accesos a 100 bloques distintos.

- Código 2: 100 accesos a bloques consecutivos, repetido 2 veces con tags diferentes. Todo esto repetido 6 veces. Genera 1200 accesos a 200 bloques distintos.
- Código 3: 100 accesos a bloques consecutivos, repetido 3 veces con tags diferentes. Todo esto repetido 6 veces. Genera 1800 accesos a 300 bloques distintos.

Estos códigos se han probado con diferentes configuraciones, comprobando que el número de aciertos para cada una de ellas es el correcto.

Para los casos donde la memoria caché es demasiado pequeña, marcados en la figura 29 en color rojo, hay 0 hits ya que, al acceder solo a 1 dirección por bloque de forma consecutiva, los bloques serán remplazados antes de ser accedidos de nuevo.

En casos donde la caché es suficientemente grande para alojar todos los bloques leídos, marcados en verde, el número de hits será el número de accesos menos los fallos iniciales.

Para los casos donde solo algunos bloques son remplazados en la caché, y marcados en amarillo, el número de aciertos es el esperado ya que, al disponer de 64 líneas por vía, e intentar cargar 100 bloques en ellas, provocando que 100-64=36 sean remplazados, solo tendremos 64-36=28 bloques que no serán remplazados provocando los 28*(6-1)=140 aciertos por vía.

Configuración / Código	1	2	3
Sin memoria caché	0	0	0
1 vía 256	140	0	0
2 vías 256	500	280	0
2 vías 512	500	1000	0
4 vías 512	500	1000	1500

Figura 29 - Test de número de hits para diferentes configuraciones.

Se puede observar como a medida que los códigos realizan más accesos a bloques diferentes en memoria, provocan más reemplazo en cachés de menor tamaño, disminuyendo el número de aciertos.

4.3 Benchmarking

A pesar de que no ha sido viable la implementación sobre la FPGA, se han llevado a cabo unas pruebas de rendimiento en simulación, con el objetivo de tener una aproximación de la mejora obtenida.

La prueba de rendimiento consistió en medir el tiempo que tarda en ejecutarse un código en específico. En nuestro caso, hemos hecho un código en ensamblador (ver figura 30) que realiza la suma de 2 vectores de 4096 elementos.

```
104   nop
105   nop
106   nop
107   la a0, A
108   li a1, 0x4
109   li a2, 0x1000 //i
110   li a3, 0x11000
111   la a4, arr
112   # li a4, 0x02340000
113   FOR1:
114       lw      t1, 0(a0) //cargamos a[i]
115       lw      t3, 0(a4) //cargamos b[i]
116       add     t2, t3, t1 //a[i]*b[i]
117       sw      t2, 0(a4)
118       add     a0, a0, a1 //actualizamos punteros
119       add     a4, a4, a1 //actualizamos punteros
120       add     a3, a3, a1 //actualizamos punteros
121       sub     a2, a2, a1 //actualizamos i
122       bgt     a2, zero, FOR1  # Repeat the loop
123   nop
124   nop
125   nop
126   nop
127   li t2, 0x01010101
128   nop
129   nop
130   nop
```

Figura 30 - Código empleado para el benchmark.

Ya que se acceden a direcciones de memoria consecutivas, la memoria caché hará que el tiempo de ejecución del programa se vea reducido en gran medida. En la gráfica de la figura 31 se puede apreciar el tiempo de ejecución con diferentes configuraciones de caché y sin caché.

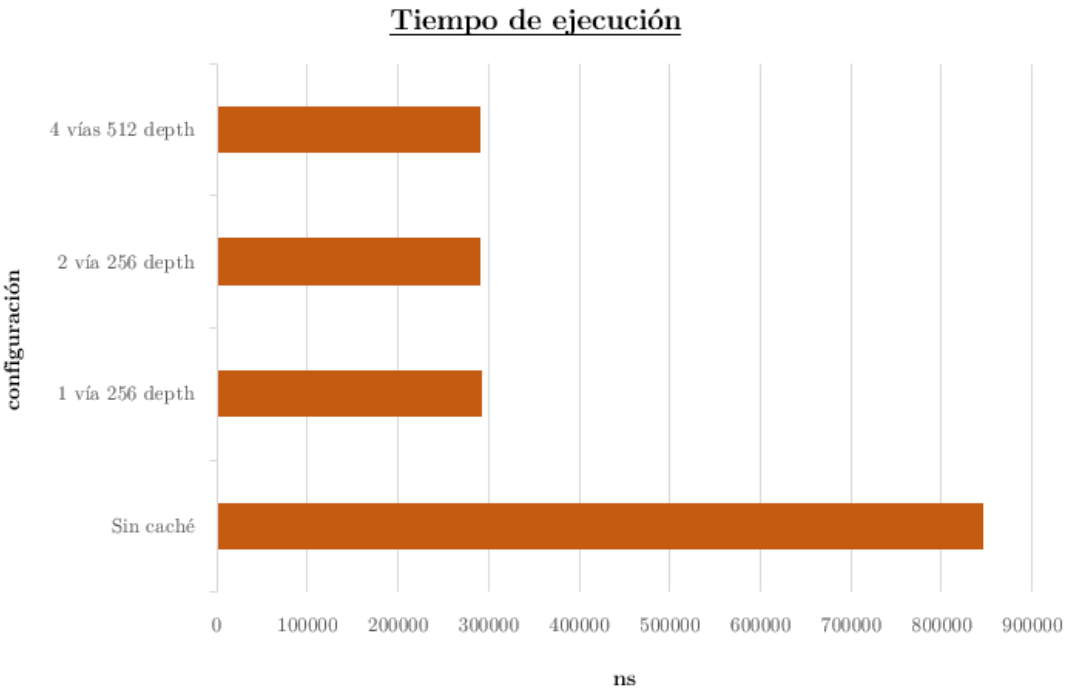

Figura 31 - Tiempo de ejecución del benchmark para diferentes configuraciones de caché.

Como podemos observar en la gráfica de la figura 31 el tiempo de ejecución para todas las configuraciones con caché es similar ya que para esta prueba ni el tamaño ni asociatividad de la caché afectan al rendimiento. Esto es debido a que el acceso a memoria de los vectores involucrados se realiza de forma secuencial y el tamaño de la memoria caché es suficiente, esto hace que la tasa de aciertos sea elevada.

El aumento de velocidad obtenido es de 2,90, no obstante, consideramos que el resultado obtenido puede ser mejorable mediante la implementación de ráfagas en el bus AXI, ya que el sobrecoste por la carga de un bloque en caché es muy alto.

Capítulo 5 Resultados y trabajo futuro

Teniendo en cuenta los objetivos del proyecto, se han cumplido gran parte de ellos. Se ha llevado a cabo el desarrollo de una memoria caché para el SweRV Core EH1, con configuración arquitectónica y políticas parametrizables. Además, se han minimizado los cambios externos a la caché de datos. Sin embargo, nuestra caché no es compatible con todas las funcionalidades existentes en el procesador, ni se puede implementar en la FPGA, debido a que no cumplimos los requerimientos temporales.

Por ello agrupamos el trabajo futuro en dos: el necesario para cumplir los objetivos del proyecto y las mejoras de diseño.

Para cumplir los **objetivos** fijados, será necesario implementar las siguientes funcionalidades.

- Cumplir los requerimientos de tiempo para la implementación en FPGA: puede ser solucionado mediante el cambio de la frecuencia de reloj de todo el procesador o la segmentación de la caché de datos.
- Soporte de ECC (error correction code) y tratamiento de errores en la paridad: actualmente el uso de ECC no está contemplado y los errores de paridad no son tratados.
- Accesos desalineados a memoria: pueden ser solucionados accediendo varias veces a memoria en estos casos.

También sería interesante implementar las siguientes **mejoras** para enriquecer el diseño y eficiencia de la caché.

- Políticas de reemplazamiento: añadir más políticas de reemplazamiento parametrizables ya que solo es soportada FIFO actualmente.
- Parametrización: ofrecer mayor nivel de parametrización, como por ejemplo diferentes tamaños de bloque.
- Secciones de memoria principal no cacheables: consideramos que esto puede ser interesante ya que ahora todo el rango de direcciones de memoria principal es cacheable.
- Mejorar las políticas de parada: parar la ruta de datos solo cuando vaya a haber una dependencia y no cuando pueda haberla. Por ejemplo, la ruta de datos se para cuándo un buffer está lleno en vez de cuando estando lleno se quiere introducir un dato nuevo.
- Non blocking loads: los loads no bloqueantes son tratados de la misma manera que los bloqueantes, paran la ruta de datos hasta recibir el dato.

- Implementación burst en AXI: implementar transferencias burst en el AXI puede ser de gran ayuda para disminuir la penalización por carga de bloque, que actualmente muy alta.

Capítulo 6 Discusión crítica

En el momento que se nos ofrece la opción de integrar una caché de datos en un procesador, pensamos que sería una tarea sencilla ya que no éramos conscientes de las implicaciones que tendría diseñar y adaptar la caché en el procesador. Sin embargo, a medida que avanzamos en el desarrollo de la caché, vimos necesario profundizar y entender mejor la arquitectura del procesador.

Inicialmente teníamos la posibilidad de buscar y adaptar una caché de código abierto al procesador, en lugar de diseñarla nosotros mismos. Optamos por el diseño con el objetivo de tener una mayor comprensión y control del funcionamiento.

Debido a esta decisión, se plantearon numerosas dificultades. Sin embargo, hemos conseguido comprender claramente el funcionamiento desempeñado por un módulo de este tipo. Además, hemos podido diseñar la caché a medida, para que el usuario pueda elegir la configuración deseada. Esto último puede ser de gran utilidad en entornos educativos, para poder observar de maneara práctica el funcionamiento a bajo nivel de una caché con diferentes configuraciones, satisfaciendo uno de nuestros objetivos principales.

Durante el desarrollo de la caché, nos surgió la oportunidad de darle un enfoque más profesional a la caché primando la fiabilidad a las opciones de configuración. Debido a los requisitos necesarios para cumplir con la propuesta, descartamos la opción ya que era necesario implementar funcionalidades que, debido al diseño realizado, conllevaría una carga de trabajo no asumible teniendo en cuenta el tiempo restante para la entrega del proyecto.

Hemos aprendido que para un diseño hardware complejo, es necesario una exhaustiva planificación previa a la codificación HDL. Si en nuestro caso, hubiésemos planificado antes de codificar el procesador, y definido objetivos más preciosos, hubiésemos evitado errores conceptuales graves durante el diseño que provocaron retrasos importantes.

También hemos aprendido que, en la práctica, un circuito está formado por un conjunto de elementos más extenso de lo que hemos visto en nuestra trayectoria universitaria. Saber cuáles de ellos son relevantes para nuestro diseño, es una parte muy importante de la fase de exploración. Adicionalmente, nunca habíamos diseñado un circuito, donde los requisitos de tiempo del reloj pudiesen llegar a ser un problema, haciéndonos ver que existen otros requisitos más complejos a los que no nos habíamos enfrentado hasta ahora en prácticas realizadas.

Hasta el momento nunca habíamos asumido el rol de diseñadores, si no que implementábamos un diseño ya existente sabiendo que era correcto. En este proyecto, esta tarea la tuvimos que desempeñar nosotros. Generando dudas de si ciertos problemas, eran causados por errores conceptuales en el diseño o la implementación.

En ocasiones teníamos que tomar decisiones que suponían un mayor rendimiento a cambio de un mayor uso de recursos hardware, pero en este aspecto nuestra experiencia era nula, por lo que era complicado mantener un equilibrio entre el rendimiento y coste.

A pesar de ello estamos satisfechos de las decisiones tomadas y del resultado obtenido.

Capítulo 7 Conclusiones

RISC-V ha supuesto una gran evolución en el mercado internacional de los procesadores, dando mucha más visibilidad al open hardware. Esta iniciativa, permite desarrollar a las empresas sus propios procesadores en vez de pagar por procesadores diseñados por otras compañías.

Western Digital está apostando por esta iniciativa con su gama de procesadores SweRV. Sin embargo, ninguno de los procesadores RISC-V abiertos implementa una caché de datos. Dada la importancia de una caché de datos en un procesador, el diseño de una es una contribución muy importante para la comunidad open hardware.

A pesar de que no hemos cumplido la totalidad de los objetivos del proyecto, hemos desarrollado una buena base sobre la que construir una caché de datos para que pueda ser publicada. Además, al ofrecer distintas configuraciones de caché es una buena herramienta para la comunidad docente.

Debido a que ha sido un proyecto muy ambicioso, el diseño ha partido de cero y han surgido dificultades con las que no contábamos en primera instancia, hemos tenido que realizar varias iteraciones para llegar al resultado final. Sin embargo, son necesarias algunas iteraciones más para cumplir todos los objetivos y conseguir un resultado más profesional.

Capítulo 8 Conclusions

RISC-V has been a great evolution in the international market for processors, giving much more presence to open hardware in this field. This initiative allows companies to develop their own, instead of paying for processors designed by other companies.

Western Digital is betting on this initiative with its range of Swerv processors. However, nowadays none of the open RISC-V processors implement a data cache. Given the importance of a data cache in a processor, designing one is a very important contribution to the open hardware community.

Although we have not fulfilled all the project objectives, we have developed a good base to build a data cache so it can be published. In addition, by offering different cache configurations, could be a good tool for the teaching community.

Because it has been a very ambitious project, the design has started from scratch and difficulties have arisen that we did not have at first, we have had to carry out several iterations to reach the final result.

However, a few more iterations are necessary to meet all our objectives and achieve a more professional result.

Contribuciones

Alfonso Carballo Boullosa

En este proyecto, y como será explicado más adelante, gran parte del trabajo ha sido realizado en conjunto mediante reuniones ya que la separación en tareas y realización de estas cada uno por su cuenta no funciona en un proyecto de este estilo. Esto es debido a que el diseño o modificación de un procesador se requiere un conocimiento muy profundo de todas las partes involucradas, tanto las añadidas (en nuestro caso la caché) como las ya existentes. Adicionalmente, en un hardware de este estilo, todas las partes esta interconectadas entre ellas de una manera u otra y siempre afectan al resto, aunque ciertas partes parezcan relativamente aisladas.

Este trabajo fue dividido en fases con un sistema de trabajo diferente para cada una de ellas.

La fase de investigación consistió, inicialmente, en la lectura de documentación oficial sobre el procesador, esta fue leída por ambos participantes. Posteriormente, puestos en común los aspectos más importantes aprendidos en una reunión.

Una vez entendida la documentación básica del procesador y con un conocimiento básico del funcionamiento de este, comenzamos la fase de exploración. La organización en esta fase fue alternada entre reuniones donde explorábamos el código e investigación individual sobre el código puesta en común más tarde. Una vez dedicado cierto tiempo a la lectura de código, nos dimos cuenta de que la comprensión mediante este método era imposible.

Como alternativa, decidimos simular el procesador y filtrar las señales más importantes según su comportamiento. Para ello, instalamos Ubuntu 20.04 y todo el software de simulación, aprendimos a simular código mediante ciertos documentos de laboratorios proporcionados por los tutores. Con el entorno de simulación instalado, ejecutamos diferentes códigos y pudimos entender cuáles eran las señales mas relevantes para la modificación, así como su comportamiento.

Debido a la elevada interconexión en un procesador de este estilo, con el fin de minimizar el tiempo de desarrollo y especialmente de depuración, solo nos dejaba una opción en cuanto al trabajo, el desarrollo conjunto del procesador.

Por lo anterior, la fase de desarrollo del procesador fue realizada mediante reuniones donde añadíamos funcionalidad a la caché y la implementábamos en el procesador. Adicionalmente y respecto a la depuración era realizada también en

conjunto, aunque en ciertas ocasiones fuese realizada por uno de los integrantes en caso de no disponer mucho tiempo para reuniones, como anteriormente este participante en la siguiente reunión resumiría los errores encontrados y las decisiones de diseño tomadas para corregirlo.

Una vez finalizada la fase de diseño, realizamos pruebas en el procesador mediante la ejecución microcódigos en ensamblador para comprobar la corrección de este en las diferentes configuraciones. Corregimos los errores que surgieron mediante el mismo procedimiento de la fase anterior.

Con el procesador con la caché de datos en su forma ya final, realizamos la prueba de rendimiento que se muestra para ver la mejora respecto al procesador inicial.

Finalmente, para la fase de escritura de esta memoria cambiamos el método de trabajo bastante, en cuanto a forma, así como herramientas usadas.

Respecto a las herramientas, inicialmente estábamos usando Google Drive para el control de versiones, documentos e información, Github para el control de versiones del código y scripts realizados. Sin embargo, para esta última fase, debido a nuestros previos conocimientos con este programa, optamos por el uso de Word. Esto hizo que, por su buena implementación, usásemos OneDrive para el almacenamiento de documentos relevantes para la memoria y Teams para las reuniones online que ahora serán explicadas.

La escritura de este documento fue (por orden de importancia) mediante reuniones online, presenciales y distribución de trabajo. El grueso del trabajo, especialmente aquellas partes más técnicas y complicadas fueron escritas en reuniones en conjunto bien online, mediante el uso de Teams y la sincronización de Word mediante OneDrive, o bien presencialmente. Sin embargo, y en contraposición al resto del trabajo, las partes más repetitivas y menos técnicamente relevantes, fueron divididas. Finalmente, aquellos aspectos de forma de documento, como los pies de foto, índice y bibliografía fueron realizados por Christian, por sus conocimientos técnicos más avanzados por el formato en Word.

Finalmente, aunque el trabajo en conjunto reduzca en parte la velocidad de desarrollo, para este caso es una muy buena opción ya que nos permite adquirir los mismos conocimientos a los dos integrantes del equipo. Adicionalmente, este estilo de trabajo nos ha permitido combinar nuestras fortalezas en diferentes campos haciendo posible que el trabajo se adaptase a nuestros altos estándares de calidad y que, con los conocimientos sesgados de solo uno de nosotros no podrían haber sido posibles.

Christian Balbás Sánchez

Debido a la falta de conocimiento para abordar este proyecto, para poder empezar a integrar la caché de datos en el procesador, fue necesario investigar acerca del procesador. Esto conllevó una lectura exhaustiva de la documentación del SweRV Core EH1 para comenzar a comprender el funcionamiento y la estructura del procesador.

Una vez comprendido esto, lo siguiente que se hizo fue abordar la comprensión del código, al menos en las partes que nos conciernen. Con el fin de comprender el significado y funcionamiento de señales fue necesario simular diferentes códigos en ensamblador para ver las variaciones de las señales ante las diferentes instrucciones.

Se llevó a cabo la instalación del entorno de simulación sobre Ubuntu 20.04, junto al software de simulación explicado en el apartado 4.1. En ocasiones el procedimiento para simular era demasiado tedioso, por lo que se llevó a cabo la creación de scripts para automatizar este proceso.

Cuando el código HDL del procesador fue comprendido más a fondo, se establecieron objetivos para ir progresando en el desarrollo de la caché.

1. Módulo que registre las direcciones accedidas.
2. Módulo que registre las direcciones accedidas y los datos.
3. Módulo que registre las direcciones accedidas y los datos para posteriormente devolverlos a la ruta de datos.

Esto conllevó el diseño de los módulos, así como la depuración para subsanar los problemas que fueron surgiendo, ayudando a comprender el flujo y funcionamiento del procesador.

Posteriormente, se decidió diseñar una caché con los módulos de RAM que se habían utilizado en la memoria caché de instrucciones, así como la estructura base de la arquitectura de la memoria. Esto hizo rediseñar la memoria caché por completo.

1. Separación del controlador de la memoria.
2. Diseño de ambos módulos para que funcionase con casos básicos de acceso a memoria, sin alterar el funcionamiento del procesador.
3. Depuración.

El diseño de la memoria caché fue condicionado por la manera en la que nos comunicamos con la memoria principal. La comunicación se hizo a través del bus AXI, lo que requirió un estudio previo del funcionamiento de las transferencias a través de este BUS. Como el bus que implementaba nuestro procesador no soportaba

la funcionalidad de burst se decidió implementar módulos lo más desacoplados posible en el controlador de la memoria, que implementan una máquina de estados para realizar una comunicación con memoria coherente y ordenada.

Una vez se consiguió un funcionamiento estable con write through y non write allocate con una caché de 4 vías y de profundidad de 256, se decidió parametrizar las políticas y la arquitectura. Lo cual requirió:

1. Rediseño basado en directivas de compilación
2. Diseño específico de políticas
3. Diseño de arquitectura parametrizable
4. Tests: probar una serie de configuraciones soportadas. Fue una tarea muy extensa, ya que había bastantes combinaciones posibles y como consecuencia de esto potenciales errores.
5. Depuración: corrección de errores en todas las configuraciones.

Con el fin de garantizar el funcionamiento del procesador con la memoria caché incorporada, se diseñaron códigos en ensamblador que ponían a prueba el funcionamiento de la memoria en casos base y extremos. Además, también se diseñaron códigos en ensamblador para poder medir la mejora respecto al procesador original.

Finalmente, se procedió a la redacción de la memoria, haciendo varias iteraciones sobre la misma para conseguir una estructura y explicación clara sobre el trabajo que hemos llevado a cabo. Esto conlleva una serie de tareas, a parte de la redacción, como por ejemplo la toma de capturas de pantalla para plasmar los códigos utilizados y la simulación de los mismos, que se repartió el trabajo equitativamente.

Todo lo anteriormente mencionado, se hizo de manera cooperativa, lo que enriqueció cada una de las fases del proyecto.

1. Investigación. se aportan gran cantidad de datos de diversa naturaleza, y se obtiene una interpretación de los mismos más rica en detalles.
2. Diseño. El hecho de que haya dos personas involucradas en el diseño de funcionalidades lo enriquece, ya que surgen nuevas ideas y facilita la prevención de problemas que puedan surgir.
3. Depuración. La corrección de errores se hace de manera más rápida y eficiente.

Bibliografía

[1] Risc-V. Cores Risc-V. Se encuentra en: https://riscv.org/exchange/cores-socs/.

[2] Risc-V. Especificaciones Risc-V. Se encuentra en: https://riscv.org/technical/specifications/

[3] Imagination. RVfpga laboratorios. Se encuentra en: https://cdn2.imgtec.com/iup-downloads/IUP_INFOSHEET_RVfpga_English.pdf

[4] Chipsalliance. Cores-SweRV https://github.com/chipsalliance/Cores-SweRV/blob/master/docs/RISC-V_SweRV_EH1_PRM.pdf

[5] ARM. Bus AXI. Se encuentra en: https://developer.arm.com/documentation/102202/0200/AXI-protocol-overview

[6] David A. Patterson, y John L. Hennessy, 2017, Computer Organization and Design RISC-V Edition, Morgan Kaufmann.